TEMA 2: ANÁLISIS LÉXICO.

El analizador léxico es el primer módulo que se utiliza, el que comienza la compilación, aunque es el analizador sintáctico el que la dirige.

El a. léxico es el responsable de encontrar las unidades y pasárselas al sintáctico: lee el programa caracter a caracter de izquierda a derecha, identifica los tokens y se los pasa al sintáctico si son léxicamente correctos. El léxico es el único módulo que lee el programa fuente. Dicho de otra forma, transforma el programa fuente en un conjunto de tokens.

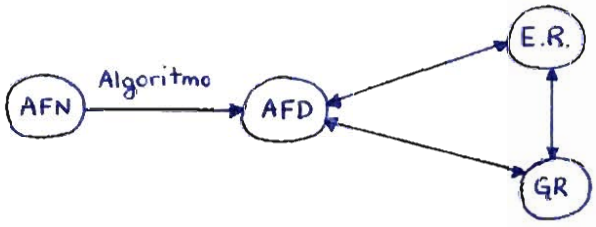
El analizador léxico detecta errores léxicas. Por ejemplo:

- Identificador demasiado largo (excede la longitud máxima permitida).
- Caracter no permitido en el lenguaje.

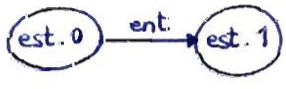
El a. léxico reconoce los tokens mediante una gramática regular o expresiones regulares.

PROPOSICIONES EQUIVALENTES:

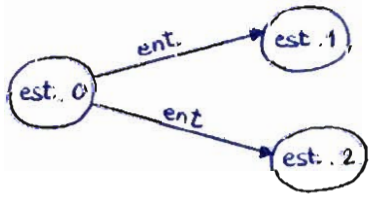
1) L es un conjunto regular.
2) L se denota por una expresión regular.
3) L puede ser definido por un AFN (autómata finito no determinista). Cualquier AFN puede ser transformado en un AFD (determinista).



Un compilador no admite: ambigüedad ⇒ Necesito un AFD ⇒ No puede ocurrir que de un paso al siguiente tenga varias opciones con la misma entrada.



AFD



AFN

No puede haber transiciones λ en el autómata.

$AF = (Q, T_e, f, q_0, F)$

- Q = Conjunto de estados del autómata.
- T_e = Alfabeto de entrada.
- f = Función de transición.
- q_0 = Estado inicial.
- F = Conjunto de estados finales.

AFD $\Rightarrow$ $f : Q \times T_e \to Q$

AFN $\Rightarrow$ $f : Q \times T_e \to P(Q)$ $\approx$ Subconjunto de Q.

El lenguaje que reconoce un autómata finito es igual al lenguaje que genera una gramática regular:

$$L(AF) = \{ t \,/\, t \in T_e^*, \ (q_0, t) \vdash^* (q_i, \lambda), \ q_i \in F \}$$

AUTÓMATA	RECONOCE	LENGUAJE.
GRAMÁTICA	GENERA	LENGUAJE.

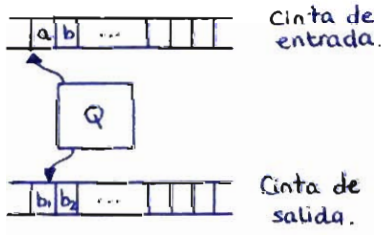
El autómata reconoce los tokens y así sé si son correctos, pero al analizador sintáctico tengo que pasarle información, decirle qué es lo que he encontrado (token + identificador). Por tanto, el analizador léxico es un Traductor Finito (TF).

El TF recibe una cadena de entrada, va transitando de un estado a otro y obtiene (traduce) la cadena de salida.

$TF = (Q, T_e, f, q_0, F)$

$TFN \Rightarrow f : Q \times (T_e \cup \lambda) \to P(Q \times T_s^*)$

$TFD \Rightarrow f : Q \times (T_e \cup \lambda) \to Q \times T_s^*$

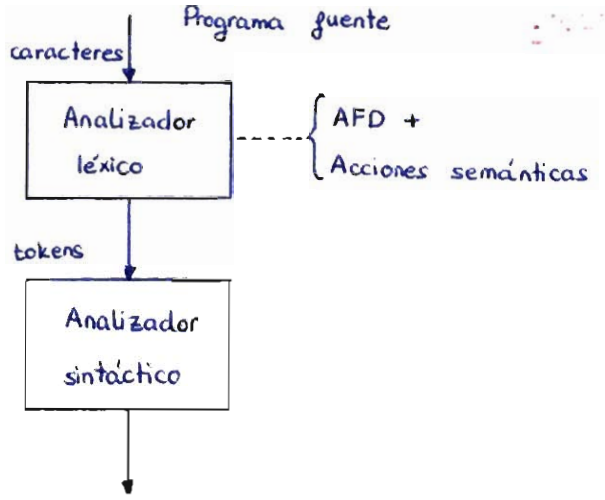


Lenguaje que traduce:

$$T_r(TF) = \{ (t, s) \,/\, t \in T_e^*, \ s \in T_s^*, \ (q_0, t, \lambda) \vdash^* (q_F, \lambda, s) \}$$

ent sal ent sal

Traducción de la cadena de entrada.



Tareas secundarias del analizador léxico:

- Maneja el fichero fuente (lo abre, lo lee, lo cierra).
- Elimina información inútil (espacios, comentarios, saltos de línea).
- Cuenta el número de línea por el que va para que, en caso de error, se pueda mostrar en qué línea se produjo).

TOKEN: Unidad mínima con información en el programa => Componente léxico mínimo con significado.

¿Cuáles son los tokens en un lenguaje de programación? Son identificadores, palabras reservadas, números (enteros, reales), operadores, signos de puntuación, constantes,...

Ejemplo: FOR i IN 1 TO MAX DO
begin

Tokens

El token es el símbolo terminal de la gramática de contexto libre con la que trabaja el analizador sintáctico.

* **PATRÓN DE UN TOKEN:** Regla que describe el conjunto de lexemas (cadenas de entrada) que podrían construir ese token (que producen una misma cadena de salida). => Regla para saber si un token está bien formado.

Los delimitadores no son tokens. Son caracteres para señalizar el final del token. Es frecuente que sea el primer caracter del token siguiente.

* **LEXEMA DE UN TOKEN:** Conjunto de caracteres del programa (cadena de entrada) que es equiparado con el patrón de un token.

Ejemplo: (Programa fuente) if cont > 5 then imprime ("Fin") ;

TOKEN	LEXEMA	PATRÓN
palabra reservada if	if, IF	if
id	cont	$l(l/d/\_)^*$ => Letra seguida de letras, números, _.
operador relacional	>	>
num_ent	5	d^+
palabra reservada then	then	then
id	imprime	$l(l/d)^*$
paréntesis abierto	(	(
literal	"Fin"	"$l(l/d)^*$"
paréntesis cerrado	)	)
punto y coma	;	;

Los números reales tienen el siguiente patrón: $d^+.d^+$ → Al menos una cifra entera y otra decimal.

Si no tengo que diferenciar los tokens de los números enteros y reales, el patrón será: $d^+[.d^+]$ (Los [] indican que es algo opcional).

¿Cómo representa / codifica el analizador léxico un token?

Cada token se representa por un par (_,_). Dicho par está formado por:

1) Primera componente => Código del token del que se trata. (palabra reservada, identificador, operador...)
2) Segunda componente => Instancia del token (lexema o posición del lexema en la tabla de símbolos). Aporta más información en los casos que sea necesario (ej: dentro de las palabras reservadas, te dice qué palabra reservada es).

Dos tipos:

TOKEN COMPLEJO: (_,_) => Se codifica como un par de números.
TOKEN SIMPLE: (_) => " " " un solo número.

Ejemplo: if cont > 5 then imprime ("Fin");

* TOKEN if. Código palabra reservada = 7 (común para todas las palabras reservadas).
Código if = 3 (dentro de las palabras reservadas, el if es el 3).

TOKEN if = (7,3)

* TOKEN cont. Código identificador = 5 (común para todos los identificadores).
Código cont = 10 (dentro de los id., el cont es el 10).

TOKEN cont: (5,10)

* TOKEN >. Código operador > = 4

TOKEN > = (4)

* TOKEN 5. Código número = 2 (común para todos los números).
Valor número = 5

TOKEN 5 = (2,5)

* TOKEN then. Código palabra reservada = 7
Código then = 4 (dentro de las palabras reservadas, el then es el 4).

TOKEN then = (7,4)

* TOKEN imprime. Código identificador = 5
Código imprime = 13

TOKEN imprime = (5, 13)

* TOKEN (. Código paréntesis = 8
Código paréntesis abierto = 1

TOKEN (= (8,1)

* TOKEN "Fin". Código literal = 3 (común para todos los literales).
Código Fin = 7 (indica que es la cadena "Fin").

TOKEN "Fin" = (3,7)

(9)

* TOKEN). Código paréntesis = 8
Código paréntesis cerrado = 2

TOKEN) = (8,2)

* TOKEN ; . Código ; = 6

TOKEN ; = (6)

Para saber estos códigos tenemos una tabla.

TOKEN	CÓD. TOKEN
palabra reservada	(7, _)
identificador	(5, _)
numero	(2, _)
op. aritmético	(9, _)
signos puntuación	
⋮	

} Estos tokens serán siempre complejos.

* TOKEN op. aritmético: (9,2)

El segundo campo es un número del 1 al 4 para poder distinguir los operadores:

+ ~ 1 * ~ 3
- ~ 2 / ~ 4

* TOKEN número: (2, _)

El segundo campo es el valor del número.

* TOKEN identificador: (5, _)

El segundo campo es:

- Lexema del identificador → Esto es así cuando el a. léxico no mete los identificadores en la TS (Tabla de Símbolos), sino que esa tarea la realiza el a. sintáctico.
- Posición del identificador en la TS → En este caso sí es el a. léxico el que mete los identificadores en la TS.

* TOKEN palabra reservada: (7, _)

El segundo campo es la posición que ocupa esa palabra reservada en una tabla, que puede ser la Tabla de Símbolos o una tabla de palabras reservadas. Si se utiliza la TS, al principio hay que inicializarla con todas las palabras reservadas. Para saber hasta dónde llegan éstas en la TS hay 2 posibilidades:

- Sé hasta dónde hay pal. reservadas en la TS.
- Existe un campo PR en la TS que marco a 1 si es pal. reservada.

Normalmente se unifica la TS (identificadores) y la tabla de pal. reservadas.

Ejercicio: (1) Reconocer y traducir un único token que sea un número entero o real sin signo.

> Nota.- Cuando me pidan el analizador léxico en un examen hay que hacer: gramática, A.F.D. y acciones semánticas.

Patrón del token que hay que reconocer: $d^+[.d^+][e[signo]d^+]$

$num \times 10^{exp}$

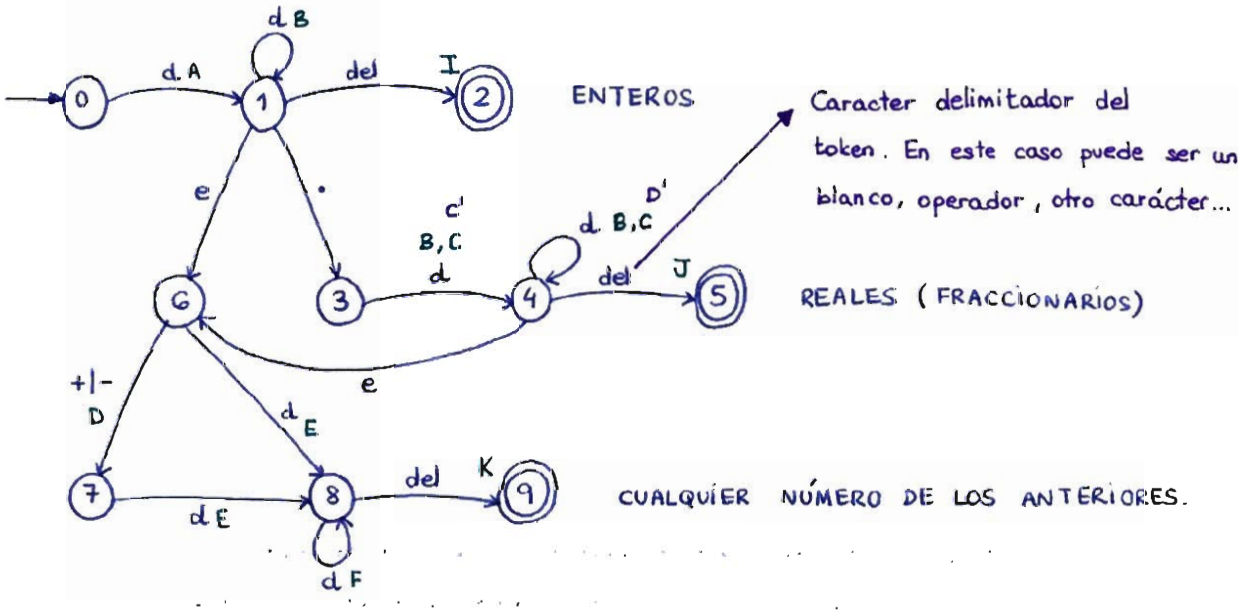
AFD => Reconocer el token, ver si es correcto.
ACCIONES SEMÁNTICAS => Traducir el token y generar el token (cod_token, valor)

Gramática regular:

N → dRN
RN → dRN | .F | eE | λ
F → dRF
RF → dRF | eE | λ
E → signo NE | dRE
NE → dRE
RE → dRE | λ
d → 0 | 1 | ... | 9
signo → + | -

En la gramática regular habría que sustituir "d" y "signo" por cada uno de sus valores posibles. "d" representa todos los dígitos.

Autómata finito determinista (AFD):



Caracter delimitador del token. En este caso puede ser un blanco, operador, otro carácter...

Acciones semánticas:

A: num := d ;
signo := +1 ;
dec := 0 ;
exp := 0 ;

B: num := num * 10 + d ;

C: dec := dec + 1 ;

D: if car = '-' then
signo = -1 ;

E: exp := exp * 10 + d ;

Si no inicializara tendría que hacer esto, pero es peor:
C': dec := 1 ;
D': dec := dec + 1 ;

En la primera transición lo mejor es inicializar todas las variables. Así no hay que hacerlo más adelante y bastará con modificar su valor cuando sea necesario.

I: gen_token (código , num) ; => Entregarlo al analizador sintáctico.
↳ número entero
" generar token

J: gen_token (código , num * 10^{-dec}) ;
↳ número real

En el estado final 9, para diferenciar si tengo un entero o real, hay que comprobar lo siguiente:

(signo * exp) - dec → si < 0 => Número real. (*)
→ si ⩾ 0 => Número entero.

⇓

K: cant := (signo * exp) - dec ;
if cant ⩾ 0 then
gen_token (código , num * 10^{cant}) ;
↳ número entero
else
gen_token (código , num * 10^{cant}) ;
↳ número real

A veces en un estado final puedo tener varios tipos de tokens que puedo distinguir mediante las acciones semánticas.

↓

Hacer "preguntas" para distinguir los tokens.

(*) Si exp. es mayor que el número de decimales, entonces es número entero
En otro caso, número real.

25 => (1 * 0) - 0 = 0 => Entero.
24'75 . 10^{1} => (1 * 1) - 2 = -1 => Real.
24'75 . 10^{-3} => (-1 * 3) - 2 = -5 => Real.

Una acción semántica puede ser muy simple o muy complicada.

Matriz de transición del AFD:

g	d.	.	e	+	-	del
0	1A					
1	1B	3	6			2I
3	4BC	(a)				
4	4BC		6			5J
6	8E	(b)		7D	7D	
7	8E		(c)			
8	8F					9K

estados no finales

(a) Punto inesperado.
(b) Punto inesperado.
(c) Formato incorrecto
} Mensajes de error.

Ejercicio: (2) Al ejercicio anterior (1) quiero añadir los operadores relacionales y la comparación:

OPERADOR	CÓDIGO
>=	0
>	1
<=	2
<	3
==	4
!=	5

Gramática regular (operadores):

$S \rightarrow > A \mid < B \mid = C \mid !D$

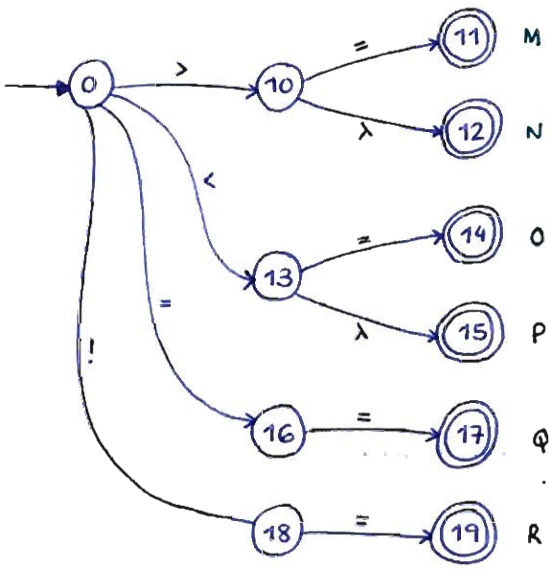
$A \rightarrow = \mid \lambda$

$B \rightarrow = \mid \lambda$

$C \rightarrow =$

$D \rightarrow =$

Autómata finito determinista (AFD): (Forma parte del AFD anterior).



Acciones semánticas:

M: gen_token (cod_op_rel, 0);
N: gen_token (cod_op_rel, 1);
O: gen_token (cod_op_rel, 2);
P: gen_token (cod_op_rel, 3);
Q: gen_token (cod_op_rel, 4);
R: gen_token (cod_op_rel, 5);

Ejercicio: Ahora también quiero reconocer el =.

(3)

OPERADOR	CÓDIGO
=	6

Gramática regular:

S → > A | < B | = C | ! D

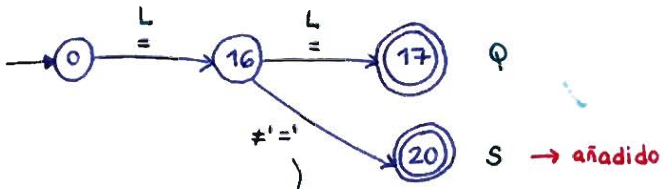
A → = | λ

B → = | λ

C → = | λ → añadido

D → =

AFD:



Acciones semánticas: → operador '=': el siguiente caracter pertenece al siguiente token (si no es un delimitador).

S: gen_token (cod_op_rel, 6);

L: Leer el siguiente caracter del fichero de entrada. Hay que usarla en el AFD cada vez que se consume un caracter.

⚠ En la transición de 16 → 20 no hay que poner la acción semántica L porque no he consumido el carácter. No lo quiero leer ahora porque lo perdería y es necesario para el siguiente token.

Al poner λ en las transiciones, tomamos λ como el conjunto de caracteres que permiten la transición en cuestión. En el caso más general representa todos los caracteres excepto el de las otras opciones de transición.

Podemos sustituir λ por "o.c." (otro caracter) o "del" (caracteres delimitadores del token).

(12)

Ejercicio: Incluir ahora:

④
- Palabras reservadas → $\ell\{\ell\}_0^\infty$ delimitador
- Identificadores → $\ell\{\ell/d\}_0^\infty$ delimitador
- Operador multiplicación → *
- Operador asignación → :=

Existen 3 posibles soluciones:

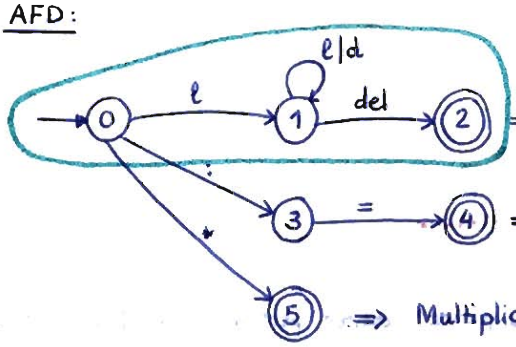
a) **Gramática regular:**

$S \to \ell R \mid * \mid :A$

$R \to \ell R \mid dR \mid \lambda$

$A \to =$

AFD:



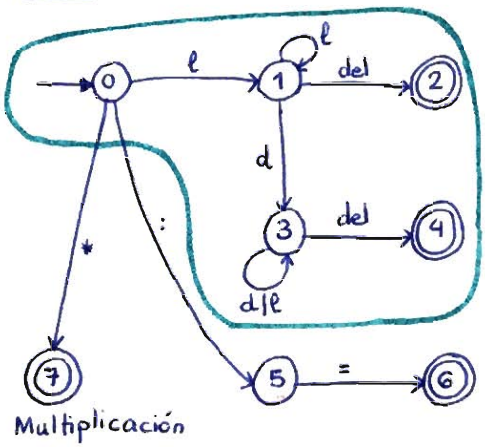
⇒ Palabra reservada o identificador.

⇒ Asignación (no necesita delimitador).

⇒ Multiplicación (no necesita delimitador).

Problema: Reconozco las palabras reservadas y los identificadores, pero no los distingo. Para diferenciarlos habría que usar acciones semánticas muy complejas.

b) **AFD:**



Multiplicación

⇒ Palabra reservada o identificador.

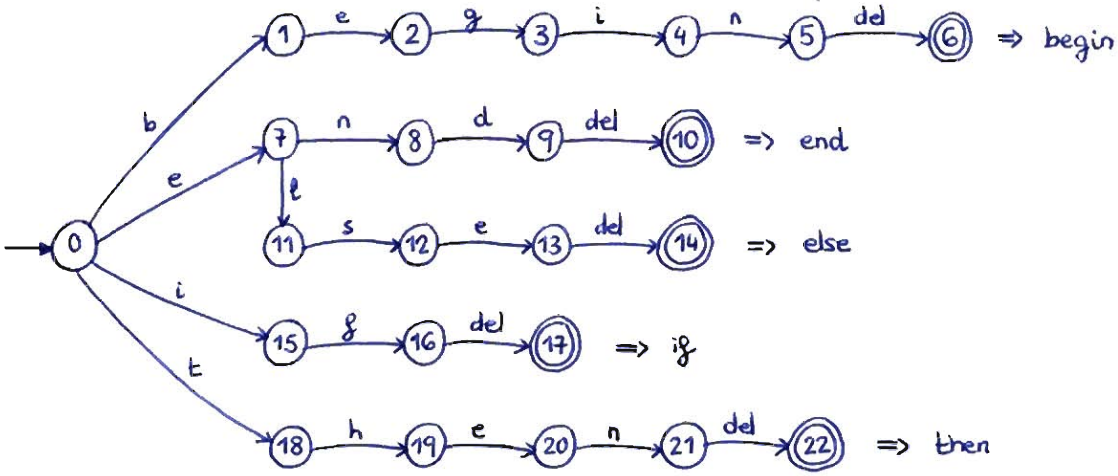
⇒ Identificador.

⇒ Asignación.

Problema: Tengo el mismo problema que antes. Aquí no los puedo distinguir si están compuestos sólo por letras. Necesito acciones semánticas para distinguirlos.

c) AFD: En esta solución detecto las palabras reservadas una por una, lo que provoca la necesidad de muchos estados.

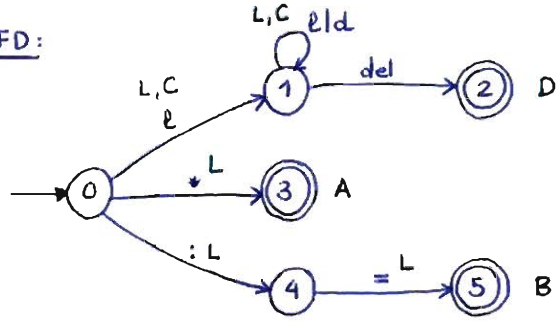
Para el caso de que todas las palabras reservadas sean: "begin, end, if, then, else" tenemos el siguiente AFD:



También tendría que reconocer los identificadores, el operador + y la asignación :=.

Solución: (a) Nos vamos a quedar con la solución a) y vamos a hacer sus acciones semánticas.

AFD:



Acciones semánticas:

A: gen_token (op_mult);

B: gen_token (op_asig);

C: Concatenar ⇒ Voy formando el lexema al concatenar lo que voy leyendo.

L: Leer el siguiente caracter del fichero de entrada.

comparando lexemas.

El compilador sabe desde el principio qué palabras son reservadas. Para diferenciar las palabras reservadas de los identificadores tengo que mirar el lugar donde las tengo almacenadas. Si la encuentro ahí, entonces sé que lo que he encontrado es una palabra reservada o, en caso contrario, es un identificador.

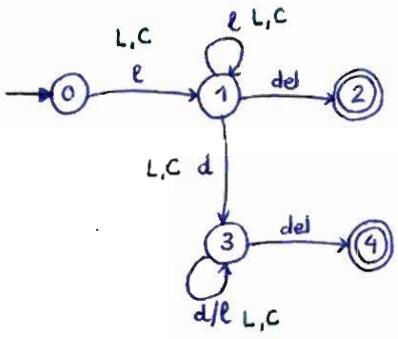
Acción semántica D => Para hacerla suponemos que el analizador léxico introduce los identificadores (con sus lexemas) en la tabla de símbolos (TS). Así, siempre que aparece un identificador, si no está en la TS lo introduzco (si estoy en la zona de declaración de variables). Las palabras reservadas están en una zona prefijada de la TS o bien hay un campo PR en la TS para identificarlas. Si tuviera 2 tablas (TS y Tabla de Palabras Reservadas) primero miraría en la TPR y si ahí no la encuentro es que se trata de un identificador).

¿Es palabra reservada?
- → SÍ: gen_token (cod_PR, código);
- → NO: Es un identificador.
  ¿Estoy en la zona de declaración de variables?
  - → SÍ: ¿Está en la tabla de símbolos?
    - → NO: gen_token (cod_identif, posición);
      Insertar en la TS.
    - → SÍ: Error => Duplicidad de variable.
  - → NO: ¿Está en la tabla de símbolos?
    - → SÍ: gen_token (cod_identif, posición);
    - → NO: Error => Identificador no declarado.

```
D:  p:= Buscar (lexema);
    if  p==0  then
        p:= Añadir (lexema);        // Identificador que no está en la TS.
    if  p == PR then
        gen_token (cod_PR, p);      // Palabra reservada.
    else
        gen_token (cod_id, p);      // Identificador que ya está en la TS.
```

Solución: Acciones semánticas para la solución b).

(b)



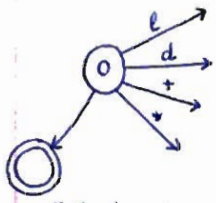
D_1 ⇒ Palabra reservada o identificador.

D_2 ⇒ Identificador.

```
D1 : p = Buscar (lexema);
     if p == 0 then
        p := Añadir (lexema);          // Identificador que no está en la TS.
     if p == PR then
        gen_token (cod_PR, p);         // Palabra reservada.
     else
        gen_token (cod_id, p);         // Identificador que ya está en la TS.
```

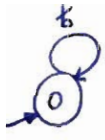
* **MENSAJES DE ERROR:** Es necesario considerar la transición de error en cualquier estado del autómata. Los mensajes de error pueden ser más o menos detallados.

Durante el análisis léxico se detectan errores, no se producen.



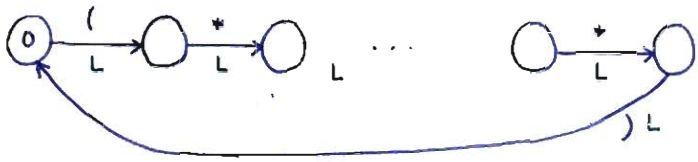
Estado de error ⇒ Si recibo algo distinto de ℓ, d, +, * entonces genero un error.

* **ESPACIOS:** ¿Qué hago con los espacios en blanco cada vez que voy a empezar a obtener un token? El autómata va eliminando estos espacios en el estado inicial.



* **COMENTARIOS:** Los comentarios no se compilan, hay que pasarlos de largo.

Ejemplo: Suponiendo que los comentarios sean de la forma (* .. *) tenemos:



En los comentarios sólo hay que hacer la acción semántica L => Sólo tengo que ir consumiendo los caracteres.

Al terminar de leer un comentario hay que volver al estado inicial.

→ El comentario se quita, pero como el a. léxico tiene que devolver un token, debe volver al estado inicial y comenzar a reconocer el próximo token.

Ejercicio: Ahora vamos a ver el autómata visto para el ejercicio 4 anterior de manera más completa.

Diseñar un analizador léxico para PR, id, := , *.

Tokens: Tenemos 4 tokens:

- PR => $\ell\{\ell\}^*$
- ID => $\ell\{\ell/d\}^*$
- OP_MULT => *
- OP_ASIG => :=

DISEÑAR UN A.L.

- Identificar tokens.
- Construir la gramática (GR).
- Construir el autómata (AFD).
- Incorporar las acciones semánticas necesarias.
- Contemplar los casos de error.
- Implementar.

Gramática:

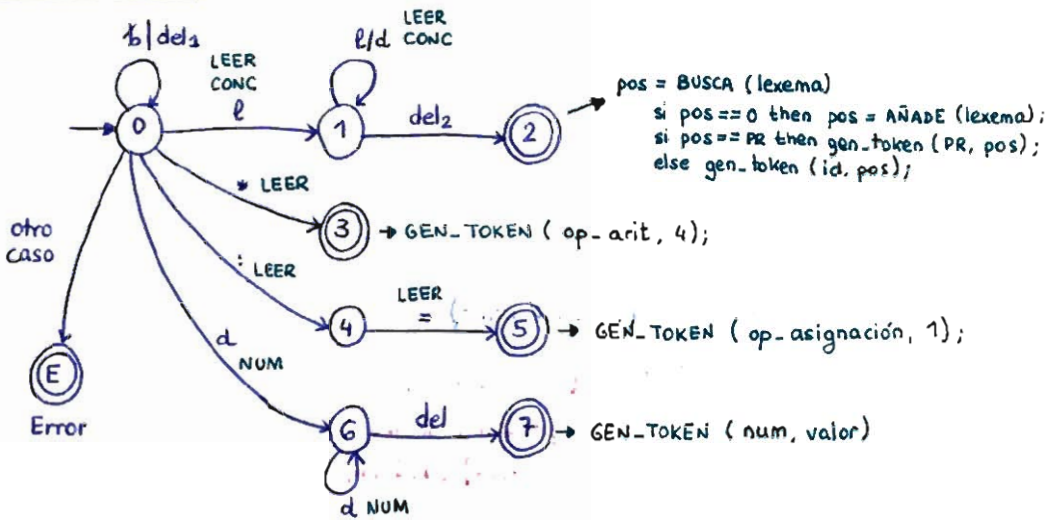
$S \rightarrow \ell R \mid * \mid : A \mid del_1 R$

$R \rightarrow \ell R \mid dR \mid del_2$

$A \rightarrow =$

Cada token tiene su propio conjunto de delimitadores.

Autómata (AFD):



Acciones semánticas:

LEER: Tomar y consumir el caracter que está a la entrada.

CONC: Construye el lexema concatenando los caracteres leídos.

GEN_TOKEN: Genera el token encontrado.

BUSCA: Busca el lexema en la TS. Devuelve la posición si lo encuentra o ∅ si no.

AÑADE: Incluye el lexema en la TS. Devuelve la posición en la que se ha insertado.

NUM: valor := valor * 10 + d;

Ejercicio: Distintos tipos de números que pueden indicarse.

Mediante los prefijos se indica la notación y mediante los sufijos el tipo.

Prefijo	Notación
nada	Decimal
0	Octal
0x	Hexadecimal

Sufijo	Tipo	Tamaño
nada	Entero normal	2
L	Entero alta precisión	4

No se pueden realizar operaciones entre constantes de distinto tipo.

Se pide diseñar un a. léxico para estos tipos de ctes.

Tokens:

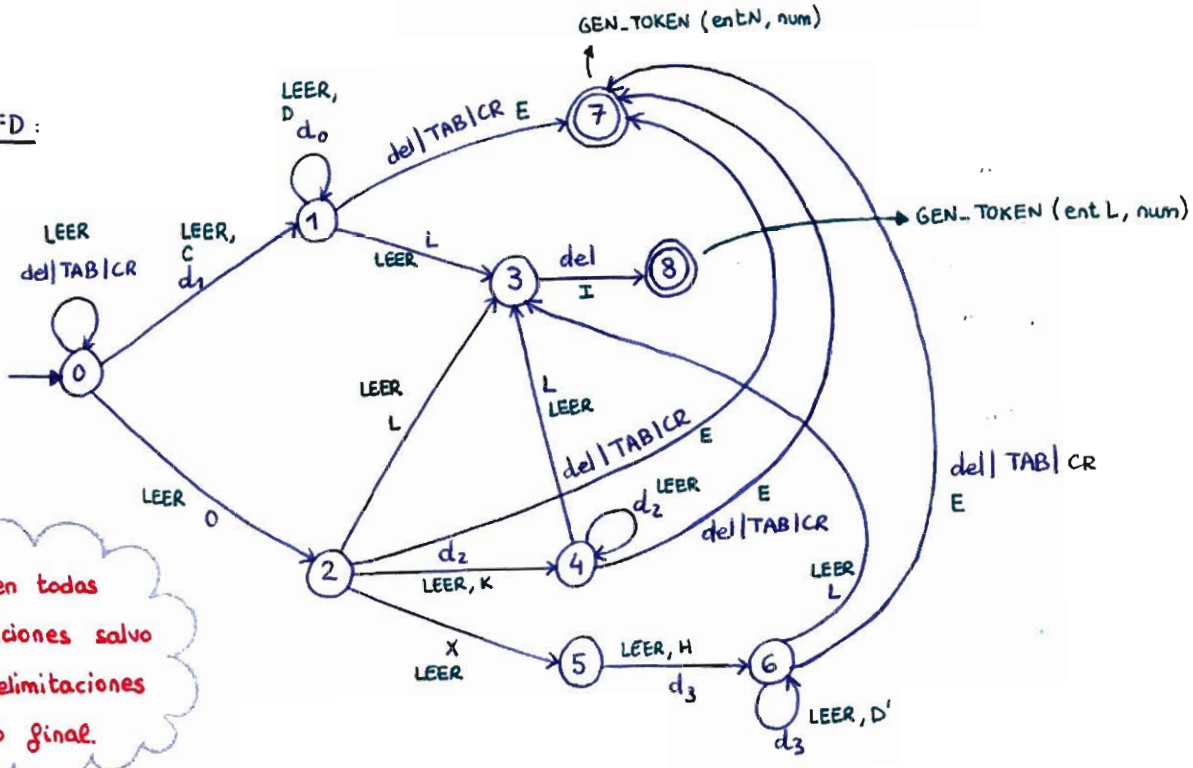
- Entero normal.
- Entero alta precisión.

Gramática regular:

$S \rightarrow d_1 A \mid 0B$
$A \rightarrow d_0 A \mid \lambda \mid L$ (terminal sufijo)
$B \rightarrow xH \mid d_2 C \mid \lambda \mid L$ (para el 0 en precisión normal y alta) (decimal)
$C \rightarrow d_2 C \mid \lambda \mid L$
$H \rightarrow d_3 E$
$E \rightarrow d_3 E \mid \lambda \mid L$

$d_1 = \{1..9\}$
$d_0 = \{0..9\}$
$d_2 = \{1..7\}$
$d_3 = \{1..9, A..F\}$

AFD:



"Leer" va en todas las transiciones salvo en las delimitaciones de estado final.

Acciones semánticas:

C: num := d;
base := 10;

D: num := num * base + d;

K: num := d_2;
base := 8;

H: num := d_3;
base := 16;

D': if car = 'A' then d = 10;
if car = 'B' then d = 11;
...
num = num * base + d;

E: if (num > 2^{16}) then error "número fuera de rango".

I: if (num > 2^{32}) then error "número fuera de rango".

Ejercicio: Tenemos que reconocer identificadores que empiezan por una letra y de longitud máxima 7.

También tenemos cadenas que empiezan y acaban con comillas.

// strings

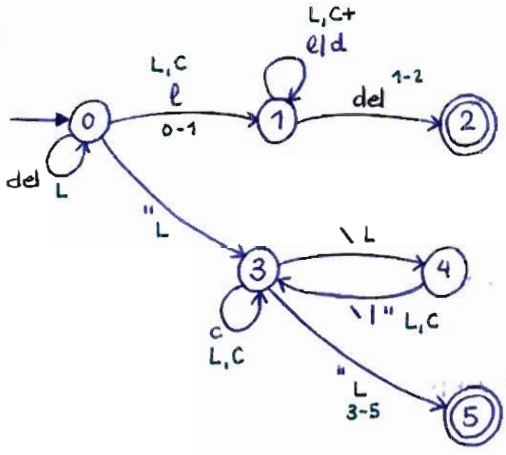
Para incluir comillas dentro de una cadena se antepone una barra invertida → \"
Para incluir la barra invertida dentro de una cadena se antepone otra barra → \\

Gramática regular:

G.R.: Símbolo terminal + [Símbolo no terminal]

A → ℓB | "C | del A
B → ℓB | dB | λ
C → cC | \D | "
D → \C | "C

AFD:



Todas las transiciones no representadas son casos de error.

gen_token (cad, p);
↓ cod_string.
↳ posición.

Acciones semánticas:

L: Leer. Avanza un caracter en la entrada.
C: Concatenar.
0-1: cant := 1;
C+: Concatenar e incrementar cantidad (cant ++)
1-2:
```
if cant ≤ 7 then {
    p := BuscarTS (pal);
    if p == Null then
        p := InsertarTS (pal);
    else
        GEN_TOKEN (id, p);
}
else
    ERROR (id_demasiado_largo);
```

3-5: p := NuevaMemoria (pal);
↓ posición
↳ string ya concatenado (y ya le hemos quitado las comillas y secuencias de escape).

Ejercicio: JUNIO 2002.

Un fragmento de un lenguaje está formado por:

- Números enteros sin signo que se representarán por 2 Bytes.
- Operadores aritméticos (+, -, *, /).
- Comentarios de dos tipos:
  - Comentarios de línea: Comienza por //. Termina por CR.
  - Comentarios de bloque: Comienza /*. Termina */

↑ Retorno de carro = Salto de línea.

Indicar todos los accesos a la tabla de símbolos.

Tokens:

- Números enteros.
- Operadores aritméticos.

Gramática regular:

A → dB | /C | del A | + | * | -

B → dB | λ (comentario de línea)

C → /D | *E | λ (división)

D → c_1 D | CR A (comentario de bloque)

E → c_2 E | * F

F → /A | c_3 E | * F

c_1 = Cualquier caracter excepto CR.

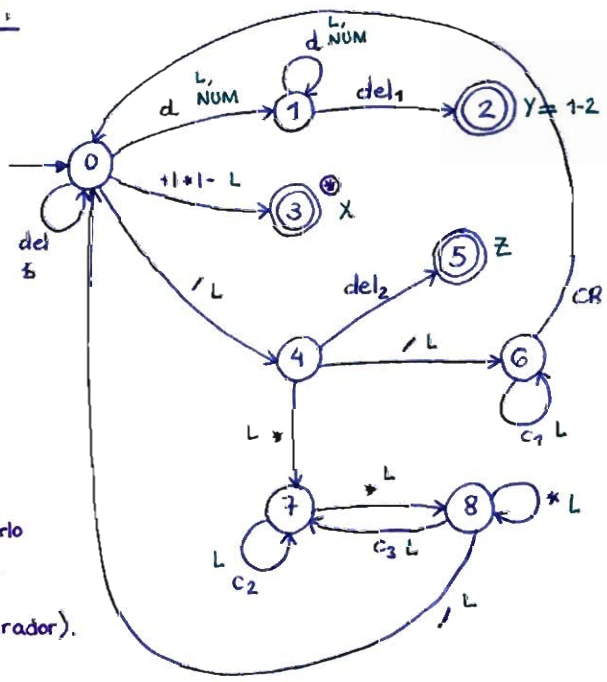
c_2 = " " " " excepto *.

c_3 = " " " " excepto / y *.

No se genera token. Se vuelve al principio.

Con los comentarios se salta programa hasta que encuentre el final del comentario.

AFD:



(*) Podría separarlo en 3 estados (uno por operador).

del_1 = Delimitador del entero.

del_2 = Cualquier caracter excepto / y *.

X: gen_token (op_arit., codigo); (+|-|*)

1-2: if num ≥ 2^{16} then (16 → 2 B = 16 bits)
  error (número fuera de rango);
else
  gen_token (entero, num);

Z: gen_token (op_arit., /);

Hablando ahora de la gestión de los identificadores, nos encontramos con que almacenar el lexema puede provocar un problema de espacio. Si en el lenguaje no hay restricción en cuanto al tamaño del lexema, ¿cómo hacemos la TS para poder almacenarlos?

Tenemos 2 opciones:

- Reservar un espacio muy grande para poder meter los identificadores muy largos => NO ES RECOMENDABLE.
- Poner en el campo de la TS un puntero hacia el identificador (hacia el vector donde están almacenados los lexemas).